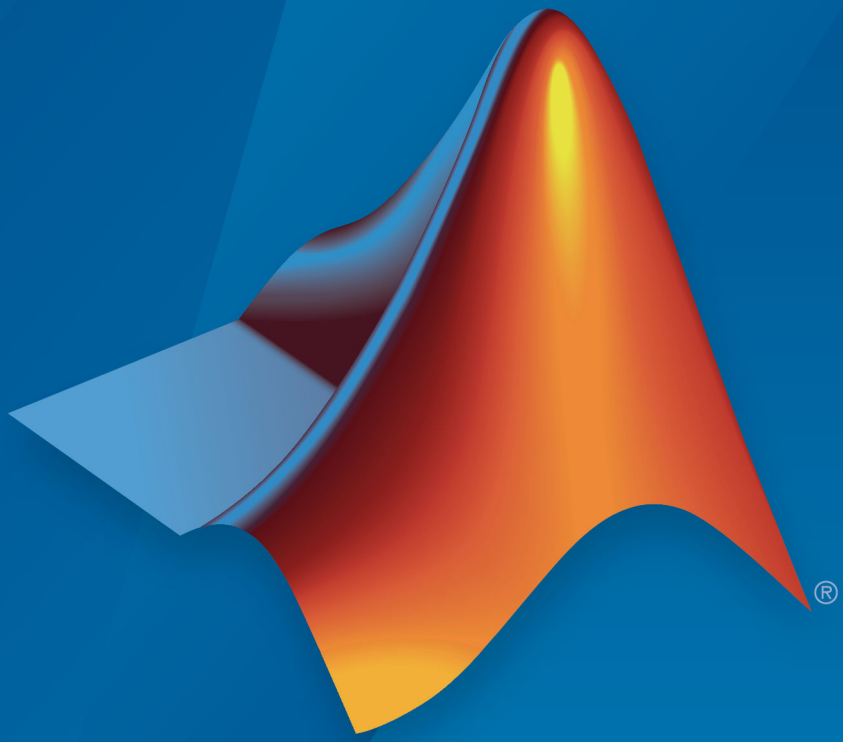
Neural Network Toolbox™ Release Notes



MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Neural Network Toolbox™ Release Notes

Contents

R2017b

R2017a

R2016b

R2016a

R2015b

R2015a

R2014b

R2014a

R2013b

R2013a

R2012b

R2012a

R2011b

R2011a

R2010b

R2010a

Bug Fixes

R2009b

Bug Fixes

R2009a

Bug Fixes

R2008b

Bug Fixes

R2008a

R2007b

R2007a

R2006b

No New Features or Changes

R2006a

R14SP3

No New Features or Changes

R2017b

Directed Acyclic Graph (DAG) Networks: Create deep learning networks with more complex architecture to improve accuracy and use many popular pretrained models

You can create and train DAG networks for deep learning. A DAG network is a neural network whose layers can be arranged as a directed acyclic graph. DAG networks can have a more complex architecture with layers that have inputs from, or outputs to, multiple layers.

To create and train a DAG network:

- Create a `LayerGraph` object using `layerGraph`. The layer graph specifies the network architecture. You can create an empty layer graph and then add layers to it. You can also create a layer graph directly from an array of network layers. The layers in the graph are automatically connected sequentially.
- Add layers to the layer graph using `addLayers` and remove layers from the graph using `removeLayers`.
- Connect layers of the layer graph using `connectLayers` and disconnect layers using `disconnectLayers`.
- Plot the network architecture using `plot`.
- Train the network using the layer graph as the layers input argument to `trainNetwork`. The trained network is a `DAGNetwork` object.
- Perform classification and prediction on new data using `classify` and `predict`.

For an example showing how to create and train a DAG network, see “Create and Train DAG Network for Deep Learning”.

You can also load a pretrained DAG network by installing the Neural Network Toolbox Model *for GoogLeNet Network* add-on. For a transfer learning example, see “Transfer Learning Using GoogLeNet”. For more information, see `googlenet`.

Long Short-Term Memory (LSTM) Networks: Create deep learning networks with the LSTM recurrent neural network topology for time-series classification and prediction

You can create and train LSTM networks for deep learning. LSTM networks are a type of recurrent neural network (RNN) that learn long-term dependencies between time steps of sequence data.

LSTM networks can be used for the following types of problems:

- Predict labels for a time series (sequence-to-label classification).
- Predict a sequence of labels for a time series (sequence-to-sequence classification).

To create an LSTM network:

- Include a sequence input layer using `sequenceInputLayer`, which inputs time-series data into the network.
- Include an LSTM layer using `lstmLayer`, which defines the LSTM architecture of the network.

For an example showing sequence-to-label classification, see “Classify Sequence Data Using LSTM Networks”.

You might want to make multiple predictions on parts of a long sequence, or might not have the complete time series in advance. For these tasks, you can make the LSTM network remember and forget the network state between predictions. To configure the state of LSTM networks, use the following functions:

- Make predictions and update the network state using `classifyAndUpdateState` and `predictAndUpdateState`.
- Reset the network state using `resetState`.

To learn more, see “Long Short-Term Memory Networks”.

Deep Learning Validation: Automatically validate network and stop training when validation metrics stop improving

You can validate deep neural networks at regular intervals during network training, and automatically stop training when validation metrics stop improving.

To perform network validation during training, specify validation data using the "'ValidationData'" name-value pair argument of `trainingOptions`. By default, the software validates the network every 50 training iterations by predicting the response of the validation data and calculating the validation loss and accuracy (root mean square error for regression networks). You can change the validation frequency using the "'ValidationFrequency'" name-value pair argument.

Network training stops when the validation loss stops improving. By default, if the validation loss is larger than or equal to the previously smallest loss five times in a row, then network training stops. To change the number of times that the validation loss is allowed to not decrease before training stops, use the "'ValidationPatience'" name-value pair argument.

For more information, see "Specify Validation Data".

Deep Learning Layer Definition: Define new layers with learnable parameters, and specify loss functions for classification and regression output layers

You can define new deep learning layers and specify your own forward propagation, backward propagation, and loss functions. To learn more, see "Define New Deep Learning Layers".

- For an example showing how to define a PReLU layer, a layer with learnable parameters, see "Define a Layer with Learnable Parameters".
- For an example showing how to define a classification output layer and specify a loss function, see "Define a Classification Output Layer".
- For an example showing how to define a regression output layer and specify a loss function, see "Define a Regression Output Layer".

Deep Learning Training Plots: Monitor training progress with plots of accuracy, loss, validation metrics, and more

You can monitor deep learning training progress by plotting various metrics during training. Plot accuracy, loss, and validation metrics to determine if and how quickly the network accuracy is improving, and whether the network is starting to overfit the training data. During training, you can stop training and return the current state of the network by clicking the stop button in the top-right corner. For example, you might want

to stop training when the accuracy of the network reaches a plateau and it is clear that the accuracy is no longer improving.

To turn on the training progress plot, use the "'Plots'" name-value pair argument of `trainingOptions`. For more information, see "Monitor Deep Learning Training Progress".

Deep Learning Image Preprocessing: Efficiently resize and augment image data for training

You can now preprocess images for network training with more options, including resizing, rotation, reflection, and other geometric transformations. To train a network using augmented images, create an `augmentedImageSource` and use it as an input argument to `trainNetwork`. You can configure augmentation options using the `imageDataAugmenter` function. For more information, see "Preprocess Images for Deep Learning".

Augmentation helps to prevent the network from overfitting and memorizing the exact details of the training images. It also increases the effective size of the training data set by generating new images based on the training images. For example, use augmentation to generate new images that randomly flip the training images along the vertical axis, and randomly translate the training images horizontally and vertically.

To resize images in other contexts, such as for prediction, classification, and network validation during training, use `imresize`.

Compatibility Considerations

In previous releases, you could perform limited image cropping and reflection using the `DataAugmentation` property of `imageInputLayer`. The `DataAugmentation` property is not recommended. Use `augmentedImageSource` instead.

Bayesian Optimization of Deep Learning: Find optimal settings for training deep networks (Requires Statistics and Machine Learning Toolbox)

Find optimal network parameters and training options for deep learning using Bayesian optimization and the `bayesopt` function. For an example, see "Deep Learning Using Bayesian Optimization".

GoogLeNet Pretrained Network: Transfer learning with pretrained GoogLeNet convolutional neural network

You can now install the Neural Network Toolbox Model *for GoogLeNet Network* add-on.

You can access the model using the `googlenet` function. If the Neural Network Toolbox Model *for GoogLeNet Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. GoogLeNet won the ImageNet Large-Scale Visual Recognition Challenge in 2014. The network is smaller and typically faster than VGG networks, and smaller and more accurate than AlexNet on the ImageNet challenge data set. The network is a directed acyclic graph (DAG) network, and `googlenet` returns the network as a `DAGNetwork` object. You can use this pretrained model for classification and transfer learning. For an example, see "Transfer Learning Using GoogLeNet". For more information on pretrained neural networks in MATLAB®, see "Pretrained Convolutional Neural Networks".

Batch Normalization Layer: Speed up network training and reduce sensitivity to network initialization

Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization. Batch normalization layers normalize the activations and gradients propagating through a neural network, making network training an easier optimization problem. To take full advantage of this fact, you can try increasing the learning rate. Because the optimization problem is easier, the parameter updates can be larger and the network can learn faster.

To create a batch normalization layer, use `batchNormalizationLayer`.

Deep Learning: New network layers

You can now use the following layers in deep learning networks:

- Batch normalization layer — Create a layer using `batchNormalizationLayer`.
- Transposed convolution layer — Create a layer using `transposedConv2dLayer`.
- Max unpooling layer — Create a layer using `maxUnpooling2dLayer`.
- Leaky Rectified Linear Unit (ReLU) layer — Create a layer using `leakyReluLayer`.
- Clipped Rectified Linear Unit (ReLU) layer — Create a layer using `clippedReluLayer`.
- Addition layer — Create a layer using `additionLayer`.
- Depth concatenation layer — Create a layer using `depthConcatenationLayer`.
- Sequence input layer for long short-term memory (LSTM) networks — Create a layer using `sequenceInputLayer`.
- LSTM layer — Create a layer using `LSTMLayer`.

Functionality Being Removed or Changed

Functionality	Result	Use Instead	Compatibility Considerations
`DataAugmentation` property of the `imageInputLayer`	Still runs	`augmentedImageSource`	The `DataAugmentation` property of `imageInputLayer` is not recommended. Use `augmentedImageSource` instead. For more information, see Deep Learning Image Preprocessing on page 1-5.

Functionality	Result	Use Instead	Compatibility Considerations
`Padding` property of `Convolution2dLayer`, `MaxPooling2dLayer`, and `AveragePooling2dLayer` objects	Warns	`PaddingSize` property of `Convolution2dLayer`, `MaxPooling2dLayer`, and `AveragePooling2dLayer` objects	Replace all instances of `Padding` property with `PaddingSize`. When you create network layers, use the `'Padding'` name-value pair argument to specify the padding. For more information, see `Convolution2dLayer`, `MaxPooling2dLayer`, and `AveragePooling2dLayer`.

R2017a

Version: 10.0

New Features

Bug Fixes

Deep Learning for Regression: Train convolutional neural networks (also known as ConvNets, CNNs) for regression tasks

You can now perform regression for numeric targets (responses) using convolutional neural networks. While defining your network, specify `regressionLayer` as the last layer. Specify the training parameters using the `trainingOptions` function. Train your network using the `trainNetwork` function. To try a regression example showing how to predict angles of rotation of handwritten digits, see Train a Convolutional Neural Network for Regression.

Pretrained Models: Transfer learning with pretrained CNN models AlexNet, VGG-16, and VGG-19, and import models from Caffe (including Caffe Model Zoo)

For pretrained convolutional neural network (CNN) models, AlexNet, VGG-16, and VGG-19, from the MATLAB Add-Ons menu, you can now install the following add-ons:

- Neural Network Toolbox™ Model *for AlexNet Network*
- Neural Network Toolbox™ Model *for VGG-16 Network*
- Neural Network Toolbox™ Model *for VGG-19 Network*

You can access the models using the functions `alexnet`, `vgg16`, and `vgg19`. These models are `SeriesNetwork` objects. You can use these pretrained models for classification and transfer learning.

You can also import other pretrained CNN models from Caffe by using the `importCaffeNetwork` function. This function imports models as a `SeriesNetwork` object. You can then use these models for classifying new data.

Alternatively, you can import CNN layers from Caffe by using the `importCaffeLayers` function. This function imports the layer architecture as a `Layer` array. You can then specify the training options using the `trainingOptions` function and train this network using the `trainNetwork` function.

For both `importCaffeNetwork` and `importCaffeLayers`, you can install the Neural Network Toolbox™ Importer *for Caffe Models* add-on from the MATLAB® Add-Ons menu.

Deep Learning with Cloud Instances: Train convolutional neural networks using multiple GPUs in MATLAB and MATLAB Distributed Computing Server for Amazon EC2

You can use MATLAB to perform deep learning in the cloud using Amazon Elastic Compute Cloud (Amazon EC2®) with new P2 instances and data stored in the cloud. If you do not have a suitable GPU available for faster training of a convolutional neural network, you can use Amazon Elastic Compute Cloud instead. Try different numbers of GPUs per machine to accelerate training. You can compare and explore the performance of multiple deep neural network configurations to find the best tradeoff of accuracy and memory use. Deep learning in the cloud also requires Parallel Computing Toolbox™. For details, see Deep Learning in the Cloud.

Deep Learning with Multiple GPUs: Train convolutional neural networks on multiple GPUs on PCs (using Parallel Computing Toolbox) and clusters (using MATLAB Distributed Computing Server)

You can now train convolutional neural networks (ConvNets) on multiple GPUs and on clusters. Specify the required hardware using the `ExecutionEnvironment` name-value pair argument in the call to the `trainingOptions` function.

Deep Learning with CPUs: Train convolutional neural networks on CPUs as well as GPUs

You can now train a convolutional neural network (ConvNet) on a CPU using the `trainNetwork` function. If there is no available GPU, by default, then `trainNetwork` uses a CPU to train the network. You can also train a ConvNet on multiple CPU cores on your desktop or a cluster using `'ExecutionEnvironment','parallel'`.

For specifying the hardware on which to train the network, and for system requirements, see the `ExecutionEnvironment` name-value pair argument on `trainingOptions`.

Deep Learning Visualization: Visualize the features ConvNet has learned using deep dream and activations

`deepDreamImage` synthesizes images that strongly activate convolutional neural network (ConvNet) layers using a version of the deep dream algorithm. Visualizing these

images highlights the features your trained ConvNet has learned, helping you understand and diagnose network behavior. For examples, see Deep Dream Images Using AlexNet and Visualize Features of a Convolutional Neural Network.

You can also display network activations on an image to investigate features the network has learned to identify. To try an example, see Visualize Activations of a Convolutional Neural Network.

table Support: Use data in tables for training of and inference with ConvNets

The `trainNetwork` function and `predict`, `activations`, and `classify` methods now accept data stored in a `table` for classification and regression problems. For details on how to specify your data, see the input argument descriptions on the function and method pages.

Progress Tracking During Training: Specify custom functions for plotting accuracy or stopping at a threshold

When training convolutional neural networks, you can specify one or more custom functions to call at each iteration during training. You can access and act on information during training, for example, to plot accuracy, or stop training early based on a threshold. Specify the functions using the `OutputFcn` name-value pair argument in `trainingOptions`. For examples, see Plot Training Accuracy During Network Training and Plot Progress and Stop Training at Specified Accuracy.

Deep Learning Examples: Get started quickly with deep learning

New examples and topics help you get started quickly with deep learning in MATLAB.

To find out what tasks you can do, see Deep Learning in MATLAB. To learn about convolutional neural networks and how they work in MATLAB, see:

- Introduction to Convolutional Neural Networks
- Specify Layers of Convolutional Neural Network
- Set Up Parameters and Train Convolutional Neural Network

New examples include:

- Try Deep Learning in 10 Lines of MATLAB Code
- Create Simple Deep Learning Network for Classification
- Transfer Learning and Fine-Tuning of Convolutional Neural Networks
- Transfer Learning Using AlexNet
- Feature Extraction Using AlexNet
- Deep Dream Images Using AlexNet
- Visualize Activations of a Convolutional Neural Network
- Visualize Features of a Convolutional Neural Network
- Create Typical Convolutional Neural Networks
- Plot Training Accuracy During Network Training
- Plot Progress and Stop Training at Specified Accuracy
- Resume Training from a Checkpoint Network
- Train a Convolutional Neural Network for Regression

R2016b

Version: 9.1

New Features

Bug Fixes

Compatibility Considerations

Deep Learning with CPUs: Run trained CNNs to extract features, make predictions, and classify data on CPUs as well as GPUs

You can choose a CPU to run a pretrained network for extracting features using `activations`, predicting image class scores using `predict`, and estimating image classes using `classify`. To specify the hardware on which to run the network, use the `'ExecutionEnvironment'` name-value pair argument in the call to the specific method.

Training a convolutional neural network (ConvNet) requires a GPU. To train a ConvNet, or to run a pretrained network on a GPU, you must have Parallel Computing Toolbox and a CUDA®-enabled NVIDIA® GPU with compute capability 3.0 or higher.

Deep Learning with Arbitrary Sized Images: Run trained CNNs on images that are different sizes than those used for training

You can run a trained convolutional neural network on arbitrary image sizes to extract features using the `activations` method with `channels` output option. For other output options, the sizes of the images you use in `activations` must be the same as the sizes of the ones used for training. To specify the `channels` output option, use the `OutputAs` name-value pair argument in the call to `activations`.

Performance: Train CNNs faster when using ImageDatastore object

`ImageDatastore` allows batch-reading of JPG or PNG image files using prefetching. This feature enables faster training of convolutional neural networks (ConvNets). If you use a custom function for reading the images, prefetching does not occur.

Deploy Training of Models: Deploy training of a neural network model via MATLAB Compiler or MATLAB Compiler SDK

Use MATLAB Runtime to deploy functions that can train a model. You can deploy MATLAB code that trains neural networks as described in Create Standalone Application from Command Line and Package Standalone Application with Application Compiler App.

The following methods and functions are NOT supported in deployed mode:

- Training progress dialog, `nntraintool`.
- `genFunction` and `gensim` to generate MATLAB code or Simulink® blocks
- `view` method
- `nctool`, `nftool`, `nnstart`, `nprtool`, `ntstool`
- Plot functions (such as `plotperform`, `plottrainstate`, `ploterrhist`, `plotregression`, `plotfit`, and so on)

generateFunction Method: generateFunction generates code for matrices by default

`'MatrixOnly'` name-value pair argument of `generateFunction` method has no effect. `generateFunction` by default generates code for only matrices.

Compatibility Considerations

You do not need to specify for `generateFunction` to generate code for matrices. Previously, you needed to specify `'MatrixOnly',true`.

alexnet Support Package: Download and use pre-trained convolutional neural network (ConvNet)

You can use pretrained Caffe version of AlexNet convolutional neural network. Download the network from the Add-Ons menu.

For more information about the network, see Pretrained Convolutional Neural Network.

R2016a

Version: 9.0

New Features

Bug Fixes

Deep Learning: Train deep convolutional neural networks with built-in GPU acceleration for image classification tasks (using Parallel Computing Toolbox)

The new functionality enables you to

- Construct convolutional neural network (CNN) architecture (see `Layer`).
- Specify training options using `trainingOptions`.
- Train CNNs using `trainNetwork` for data in 4D arrays or `ImageDatastore`.
- Make predictions of class labels using a trained network using `predict` or `classify`.
- Extract features from a trained network using `activations`.
- Perform transfer learning. That is, retrain the last fully connected layer of an existing CNN on new data.

NOTE: This feature requires the Parallel Computing Toolbox and a CUDA-enabled NVIDIA GPU with compute capability 3.0 or higher.

R2015b

Version: 8.4

New Features

Bug Fixes

Autoencoder neural networks for unsupervised learning of features using the trainAutoencoder function

You can train autoencoder neural networks to learn features using the `trainAutoencoder` function. The trained network is an `Autoencoder` object. You can use the trained autoencoder to predict the inputs for new data, using the `predict` method. For all the properties and methods of the object, see the `Autoencoder` class page.

Deep learning using the stack function for creating deep networks from autoencoders

You can create deep networks using the `stack` method. To create a deep network, after training the autoencoders, you can

1. Extract features from autoencoders using the `encode` method.
2. Train a softmax layer for classification using the `trainSoftmaxLayer` function.
3. Stack the encoders and the softmax layer to form a deep network, and train the deep network.

The deep network is a `network` object.

Improved speed and memory efficiency for training with Levenberg-Marquardt (trainlm) and Bayesian Regularization (trainbr) algorithms

An optimized MEX version of the Jacobian backpropagation algorithm allows faster training and reduces memory requirements for training static and open-loop networks using the `trainlm` and `trainbr` functions.

Cross entropy for a single target variable

The `crossentropy` function supports binary encoding, that is, when there are only two classes and N = 1 (N is the number of rows in the `targets` input argument).

R2015a

Version: 8.3

New Features

Bug Fixes

Progress update display for parallel training

The Neural Network Training tool (`nntraintool`) now displays progress updates when conducting parallel training of a network.

R2014b

Version: 8.2.1

Bug Fixes

R2014a

Version: 8.2

New Features

Bug Fixes

Training panels for Neural Fitting Tool and Neural Time Series Tool Provide Choice of Training Algorithms

The training panels in the Neural Fitting and Neural Time Series tools now let you select a training algorithm before clicking **Train**. The available algorithms are:

- Levenberg-Marquardt (`trainlm`)
- Bayesian Regularization (`trainbr`)
- Scaled Conjugate Gradient (`trainscg`)

For more information on using Neural Fitting, see Fit Data with a Neural Network.

For more information on using Neural Time Series, see Neural Network Time Series Prediction and Modeling.

Bayesian Regularization Supports Optional Validation Stops

Because Bayesian-Regularization with `trainbr` can take a long time to stop, validation used with Bayesian-Regularization allows it to stop earlier, while still getting some of the benefits of weight regularization. Set the training parameter `trainParam.max_fail` to specify when to make a validation stop. Validation is disabled for `trainbr` by default when `trainParam.max_fail` is set to 0.

For example, to train as before without validation:

```
[x,t] = house_dataset;
net = feedforwardnet(10,'trainbr');
[net,tr] = train(net,x,t);
```

To train with validation:

```
[x,t] = house_dataset;
net = feedforwardnet(10,'trainbr');
net.trainParam.max_fail = 6;
[net,tr] = train(net,x,t);
```

Neural Network Training Tool Shows Calculations Mode

Neural Network Training Tool now shows its calculations mode (i.e., MATLAB, GPU) in its **Algorithms** section.

R2013b

Function code generation for application deployment of neural network simulation (using MATLAB Coder, MATLAB Compiler, and MATLAB Builder products)

New Function: genFunction

The function `genFunction` generates a stand-alone MATLAB function for simulating any trained neural network and preparing it for deployment in many scenarios:

- Document the input-output transforms of a neural network used as a calculation template for manual reimplementations of the network
- Create a Simulink block using the MATLAB Function block
- Generate C/C++ code with MATLAB Coder™ `codegen`
- Generate efficient MEX-functions with MATLAB Coder `codegen`
- Generate stand-alone C executables with MATLAB Compiler™ `mcc`
- Generate C/C++ libraries with MATLAB Compiler `mcc`
- Generate Excel® and .COM components with MATLAB Builder™ EX `mcc` options
- Generate Java components with MATLAB Builder JA `mcc` options
- Generate .NET components with MATLAB Builder NE `mcc` options

`genFunction(net,'path/name')` takes a neural network and file path and produces a standalone MATLAB function file `'name.m'`.

`genFunction(_____,'MatrixOnly','yes')` overrides the default cell/matrix notation and instead generates a function that uses only matrix arguments compatible with MATLAB Coder tools. For static networks the matrix columns are interpreted as independent samples. For dynamic networks the matrix columns are interpreted as a series of time steps. The default value is `'no'`.

`genFunction(_____,'ShowLinks','no')` disables the default behavior of displaying links to generated help and source code. The default is `'yes'`.

Here a static network is trained and its outputs calculated.

```
[x,t] = house_dataset;
houseNet = feedforwardnet(10);
houseNet = train(houseNet,x,t);
y = houseNet(x);
```

A MATLAB function with the same interface as the neural network object is generated and tested, and viewed.

```
genFunction(houseNet,'houseFcn');
y2 = houseFcn(x);
accuracy2 = max(abs(y-y2))
edit houseFcn
```

The new function can be compiled with the MATLAB Compiler tools (license required) to a shared/dynamically linked library with `mcc`.

```
mcc -W lib:libHouse -T link:lib houseFcn
```

Next, another version of the MATLAB function is generated which supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool codegen (license required) which is also tested.

```
genFunction(houseNet,'houseFcn','MatrixOnly','yes');
y3 = houseFcn(x);
accuracy3 = max(abs(y-y3))

x1Type = coder.typeof(double(0),[13 Inf]); % Coder type of input 1
codegen houseFcn.m -config:mex -o houseCodeGen -args {x1Type}
y4 = houseCodeGen(x);
accuracy4 = max(abs(y-y4))
```

Here, a dynamic network is trained and its outputs calculated.

```
[x,t] = maglev_dataset;
maglevNet = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(maglevNet,x,{},t);
maglevNet = train(maglevNet,X,T,Xi,Ai);
[y,xf,af] = maglevNet(X,Xi,Ai);
```

Next, a MATLAB function is generated and tested. The function is then used to create a shared/dynamically linked library with `mcc`.

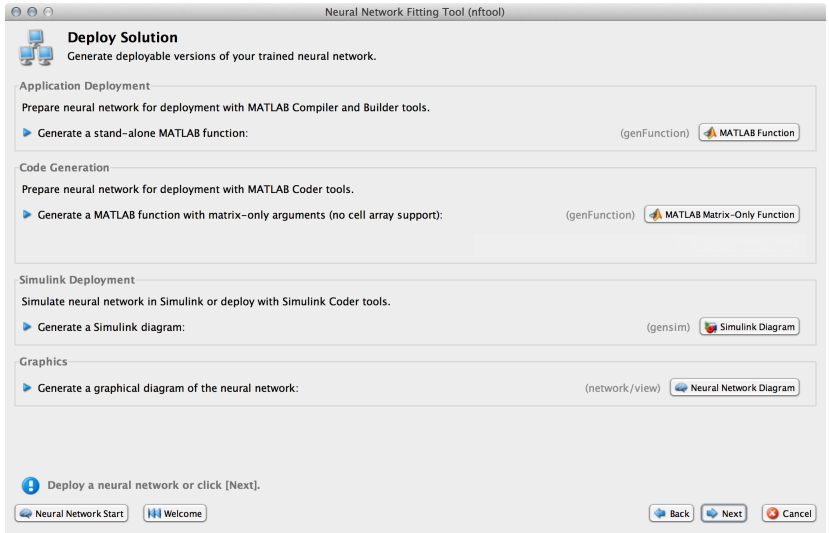
```
genFunction(maglevNet,'maglevFcn');
[y2,xf,af] = maglevFcn(X,Xi,Ai);
accuracy2 = max(abs(cell2mat(y)-cell2mat(y2)))
mcc -W lib:libMaglev -T link:lib maglevFcn
```

Next, another version of the MATLAB function is generated which supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool `codegen`, and the result is also tested.

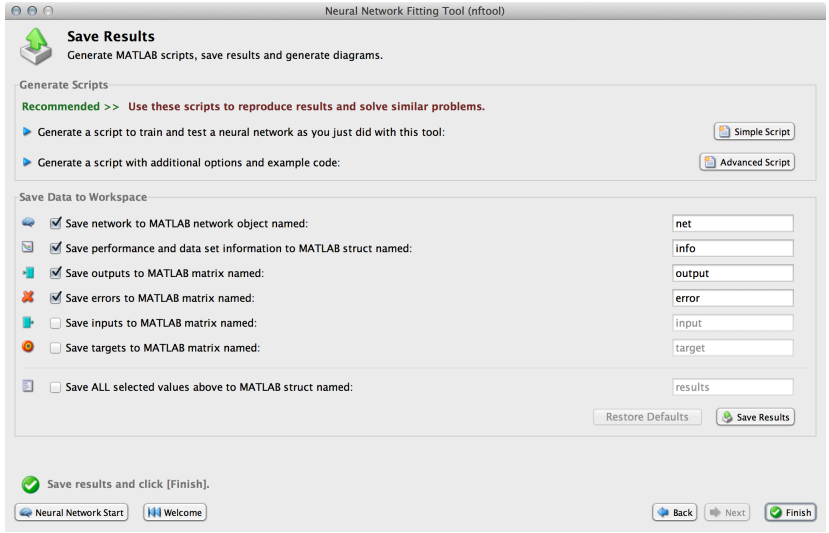
```
genFunction(maglevNet,'maglevFcn','MatrixOnly','yes');
x1 = cell2mat(X(1,:)); % Convert each input to matrix
x2 = cell2mat(X(2,:));
xi1 = cell2mat(Xi(1,:)); % Convert each input state to matrix
xi2 = cell2mat(Xi(2,:));
[y3,xf1,xf2] = maglevFcn(x1,x2,xi1,xi2);
accuracy3 = max(abs(cell2mat(y)-y3))

x1Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 1
x2Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 2
xi1Type = coder.typeof(double(0),[1 2]); % Coder type of input 1 states
xi2Type = coder.typeof(double(0),[1 2]); % Coder type of input 2 states
codegen maglevFcn.m -config:mex -o maglevNetCodeGen -args {x1Type x2Type xi1Type xi2Type}
[y4,xf1,xf2] = maglevNetCodeGen(x1,x2,xi1,xi2);
dynamic_codegen_accuracy = max(abs(cell2mat(y)-y4))
```

Enhanced Tools

The function `genFunction` is introduced with a new panel in the tools `nftool`, `nctool`, `nprtool` and `ntstool`.



The advanced scripts generated on the Save Results panel of each of these tools includes an example of deploying networks with `genFunction`.



For more information, see Deploy Neural Network Functions.

Enhanced multi-timestep prediction for switching between open-loop and closed-loop modes with NARX and NAR neural networks

Dynamic networks with feedback, such as `narxnet` and `narnet` neural networks, can be transformed between open-loop and closed-loop modes with the functions `openloop` and `closeloop`. Closed-loop networks make multistep predictions. In other words, they continue to predict when external feedback is missing, by using internal feedback.

It can be useful to simulate a trained neural network up the present with all the known values of a time-series in open-loop mode, then switch to closed-loop mode to continue the simulation for as many predictions into the future as are desired. It is now much easier to do this.

Previously, `openloop` and `closeloop` transformed the neural network between those two modes.

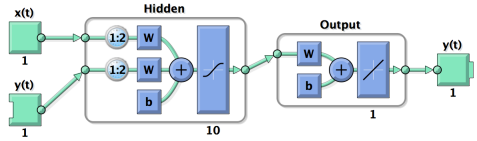
```
net = openloop(net)
net = closeloop(net)
```

This is still the case. However, these functions now also support the transformation of input and layer delay state values between open- and closed-loop modes, making switching between closed-loop to open-loop multistep prediction easier.

```
[net,xi,ai] = openloop(net,xi,ai);
[net,xi,ai] = closeloop(net,xi,ai);
```

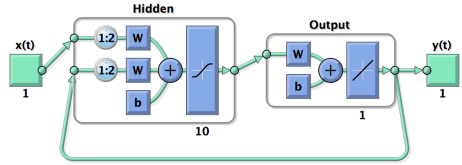
Here, a neural network is trained to model the magnetic levitation system in default open-loop mode.

```
[X,T] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
view(net)
```



Then `closeloop` is used to convert the network to closed-loop form for simulation.

```
netc = closeloop(net);
[x,xi,ai,t] = preparets(netc,X,{},T);
y = netc(x,xi,ai);
view(netc)
```



Now consider the case where you might have a record of the Maglev's behavior for 20 time steps, but then want to predict ahead for 20 more time steps beyond that.

Define the first 20 steps of inputs and targets, representing the 20 time steps where the output is known, as defined by the targets `t`. Then the next 20 time steps of the input are defined, but you use the network to predict the 20 outputs using each of its predictions feedback to help the network perform the next prediction.

```
x1 = x(1:20);
t1 = t(1:20);
x2 = x(21:40);
```

Then simulate the open-loop neural network on this data:

```
[x,xi,ai,t] = preparets(net,x1,{},t1);
[y1,xf,af] = net(x,xi,ai);
```

Now the final input and layer states returned by the network are converted to closed-loop form along with the network. The final input states `xf`, and layer states `af`, of the open-loop network become the initial input states `xi`, and layer states `ai`, of the closed-loop network.

```
[netc,xi,ai] = closeloop(net,xf,af);
```

Typically, `preparets` is used to define initial input and layer states. Since these have already been obtained from the end of the open-loop simulation, you do not need `preparets` to continue with the 20 step predictions of the closed-loop network.

```
[y2,xf,af] = netc(x2,xi,ai);
```

Note that `x2` can be set to different sequences of inputs to test different scenarios for however many time steps you would like to make predictions. For example, to predict the magnetic levitation system's behavior if 10 random inputs were used:

```
x2 = num2cell(rand(1,10));
[y2,xf,af] = netc(x2,xi,ai);
```

For more information, see Multistep Neural Network Prediction.

Cross-entropy performance measure for enhanced pattern recognition and classification accuracy

Networks created with `patternnet` now use the cross-entropy performance measure (`crossentropy`), which frequently produces classifiers with fewer percentage misclassifications than obtained using mean squared error.

See "Softmax transfer function in output layer gives consistent class probabilities for pattern recognition and classification" on page 9-8.

Softmax transfer function in output layer gives consistent class probabilities for pattern recognition and classification

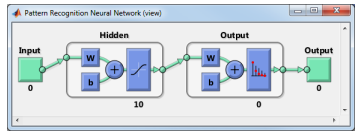
`patternnet`, which you use to create a neural network suitable for learning classification problems, has been improved in two ways.

First, networks created with `patternnet` now use the cross-entropy performance measure (`crossentropy`), which frequently produces classifiers with fewer percentage misclassifications than obtained using mean squared error.

Second, `patternnet` returns networks that use the Soft Max transfer function (`softmax`) for the output layer instead of the `tansig` sigmoid transfer function. `softmax` results in output vectors normalized so they sum to 1.0, that can be interpreted as class probabilities. (`tansig` also produces outputs in the 0 to 1 range, but they do not sum to 1.0 and have to be manually normalized before being treated as consistent class probabilities.)

Here a `patternnet` with 10 neurons is created, its performance function and diagram are displayed.

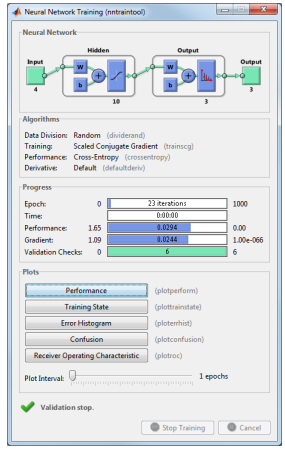
```
net = patternnet(10);
net.performFcn

ans =
crossentropy

view(net)
```



The output layer's transfer function is shown with the symbol for `softmax`.

Training the network takes advantage of the new `crossentropy` performance function. Here the network is trained to classify iris flowers. The cross-entropy performance algorithm is shown in the `nntraintool` algorithm section. Clicking the "Performance" plot button shows how the network's cross-entropy was minimized throughout the training session.

```
[x,t] = iris_dataset;
net = train(net,x,t);
```



Simulating the network results in normalized output. Sample 150 is used to illustrate the normalization of class membership likelihoods:

```
y = net(x(:,150))
```

```
y  =

    0.0001
    0.0528
    0.9471
```

```
sum(y)
```

```
     1
```

The network output shows three membership probabilities with class three as by far the most likely. Each probability value is between 0 and 1, and together they sum to 1 indicating the 100% probability that the input `x(:,150)` falls into one of the three classes.

Compatibility Considerations

If a `patternnet` network is used to train on target data with only one row, the network's output transfer function will be changed to `tansig` and its outputs will continue to

operate as they did before the `softmax` enhancement. However, the `1-of-N` notation for targets is recommended even when there are only two classes. In that case the targets should have two rows, where each column has a `1` in the first or second row to indicate class membership.

If you prefer the older `patternnet` of mean squared error performance and a sigmoid output transfer function, you can specify this by setting those neural network object properties. Here is how that is done for a `patternnet` with 10 neurons.

```
net = patternnet(10);
net.layers{2}.transferFcn = 'tansig';
net.performFcn = 'mse';
```

Automated and periodic saving of intermediate results during neural network training

Intermediate results can be periodically saved during neural network training to a `.mat` file for recovery if the computer fails or the training process is killed. This helps protect the values of long training runs, which if interrupted, would otherwise need to be completely restarted.

This feature can be especially useful for long parallel training sessions that are more likely to be interrupted by computing resource failures and which you can stop only with a Ctrl+C break, because the `nntraintool` tool (with its **Stop** button) is not available during parallel training.

Checkpoint saves are enabled with an optional `'CheckpointFile'` training argument followed by the checkpoint file's name or path. If only a file name is specified, it is placed in the current folder by default. The file must have the `.mat` file extension, but if it is not specified it is automatically added. In this example, checkpoint saves are made to a file called `MyCheckpoint.mat` in the current folder.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net2 = train(net,x,t,'CheckpointFile','MyCheckpoint.mat');

22-Mar-2013 04:49:05 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:49:06 Final Checkpoint #2: /WorkingDir/MyCheckpoint.mat
```

By default, checkpoint saves occur at most once every 60 seconds. For the short training example above this results in only two checkpoints, one at the beginning and one at the end of training.

The optional training argument `'CheckpointDelay'` changes the frequency of saves. For example, here the minimum checkpoint delay is set to 10 seconds, for a time-series problem where a neural network is trained to model a levitated magnet.

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);

22-Mar-2013 04:59:28 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:38 Write Checkpoint #2: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:48 Write Checkpoint #3: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:58 Write Checkpoint #4: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:08 Write Checkpoint #5: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:09 Final Checkpoint #6: /WorkingDir/MyCheckpoint.mat
```

After a computer failure or training interruption, the checkpoint structure containing the best neural network obtained before the interruption and the training record can be reloaded. In this case the `stage` field value is `'Final'`, indicating the last save was at the final epoch, because training completed successfully. The first epoch checkpoint is indicated by `'First'`, and intermediate checkpoints by `'Write'`.

```
load('MyCheckpoint.mat')

checkpoint =

      file: '/WorkingDir/MyCheckpoint.mat'
      time: [2013 3 22 5 0 9.0712]
    number: 6
     stage: 'Final'
       net: [1x1 network]
        tr: [1x1 struct]
```

Training can be resumed from the last checkpoint by reloading the dataset (if necessary), then calling `train` with the recovered network.

```
net = checkpoint.net;
[x,t] = maglev_dataset;
load('MyCheckpoint.mat');
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);
```

For more information, see Automatically Save Checkpoints During Neural Network Training.

Simpler Notation for Networks with Single Inputs and Outputs

The majority of neural networks have a single input and single output. You can now refer to the input and output of such networks with the properties `net.input` and `net.output`, without the need for cell array indices.

Here a feed-forward neural network is created and its input and output properties examined.

```
net = feedforwardnet(10);
net.input
net.output
```

The `net.inputs{1}` notation for the input and `net.outputs{2}` notation for the second layer output continue to work. The cell array notation continues to be required for networks with multiple inputs and outputs.

For more information, see Neural Network Object Properties.

Neural Network Efficiency Properties Are Now Obsolete

The neural network property `net.efficiency` is no longer shown when a network object properties are displayed. The following line of code displays the properties of a feed-forward network.

```
net = feedforwardnet(10)
```

Compatibility Considerations

The efficiency properties are still supported and do not yet generate warnings, so backward compatibility is maintained. However the recommended way to use memory reduction is no longer to set `net.efficiency.memoryReduction`. The recommended notation since R2012b is to use optional training arguments:

```
[x,t] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,x,t,'Reduction',10);
```

Memory reduction is a way to trade off training time for lower memory requirements when using Jacobian training such as `trainlm` and `trainbr`. The `MemoryReduction` value indicates how many passes must be made to simulate the network and calculate its

gradients each epoch. The storage requirements go down as the memory reduction goes up, although not necessarily proportionally. The default `MemoryReduction` is 1, which indicates no memory reduction.

R2013a

Version: 8.0.1

Bug Fixes

R2012b

Version: 8.0

New Features

Bug Fixes

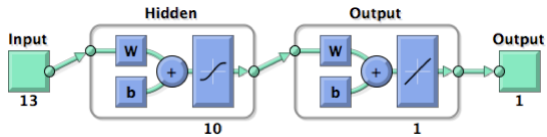
Compatibility Considerations

Speed and memory efficiency enhancements for neural network training and simulation

The neural network simulation, gradient, and Jacobian calculations are reimplemented with native MEX-functions in Neural Network Toolbox Version 8.0. This results in faster speeds, especially for small to medium network sizes, and for long time-series problems.

In Version 7, typical code for training and simulating a feed-forward neural network looks like this:

```
[x,t] = house_dataset;
net = feedforwardnet(10);
view(net)
net = train(net,x,t);
y = net(x);
```



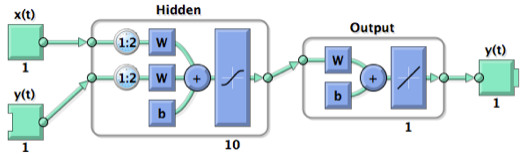
In Version 8.0, the above code does not need to be changed, but calculations now happen in compiled native MEX code.

Speedups of as much as 25% over Version 7.0 have been seen on a sample system (4-core 2.8 GHz Intel i7 with 12 GB RAM).

Note that speed improvements measured on the sample system might vary significantly from improvements measured on other systems due to different chip speeds, memory bandwidth, and other hardware and software variations.

The following code creates, views, and trains a dynamic NARX neural network model of a maglev system in open-loop mode.

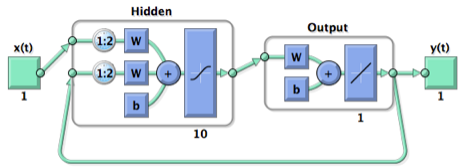
```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
view(net)
[X,Xi,Ai,T] = preparets(net,x,{},t);
net = train(net,X,T,Xi,Ai);
y = net(X,Xi,Ai)
```



The following code measures training speed over 10 training sessions, with the training window disabled to avoid GUI timing interference.

On the sample system, this ran three times (3x) faster in Version 8.0 than in Version 7.0.

```
rng(0)
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,{},t);
net.trainParam.showWindow = false;
tic
for i=1:10
  net = train(net,X,T,Xi,Ai);
end
toc
```



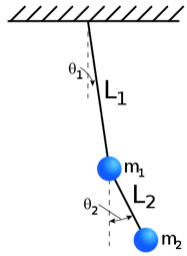
The following code trains the network in closed-loop mode:

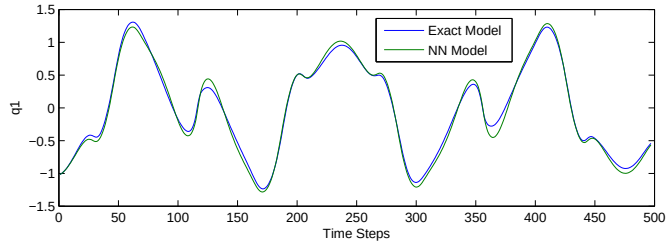
```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
net = closeloop(net);
view(net)
[X,Xi,Ai,T] = preparets(net,x,{},t);
net = train(net,X,T,Xi,Ai);
```

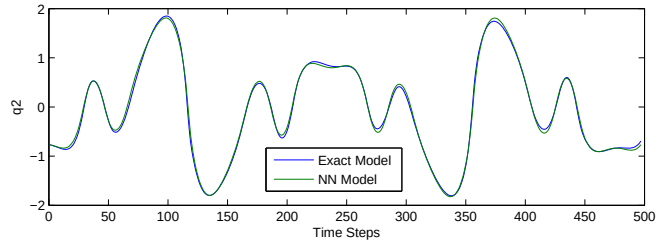
For this case, and most closed-loop (recurrent) network training, Version 8.0 ran the code more than one-hundred times (100x) faster than Version 7.0.

A dramatic example of where the improved closed loop training speed can help is when training a NARX network model of a double pendulum. By initially training the network

in open-loop mode, then in closed-loop mode with two time step sequences, then three time step sequences, etc., a network has been trained that can simulate the system for 500 time steps in closed-loop mode. This corresponds to a 500 step ahead prediction.







Because of the Version 8.0 MEX speedup, this only took a few hours, as apposed to the months it would have taken in Version 7.0.

MEX code is also far more memory efficient. The amount of RAM used for intermediate variables during training and simulation is now relatively constant, instead of growing linearly with the number of samples. In other words, a problem with 10,000 samples requires the same temporary storage as a problem with only 100 samples.

This memory efficiency means larger problems can be trained on a single computer.

Compatibility Considerations

For very large networks, MEX code might fall back to MATLAB code. If this happens and memory availability becomes an issue, use the `'reduction'` option to implement memory reduction. The reduction number indicates the number of passes to make through the data for each calculation. Each pass calculates with a fraction of the data, and the results are combined after all passes are complete. This trades off lower memory requirements for longer calculation times.

```
net = train(net,x,t,'reduction',10);
y = net(x,'reduction',10);
```

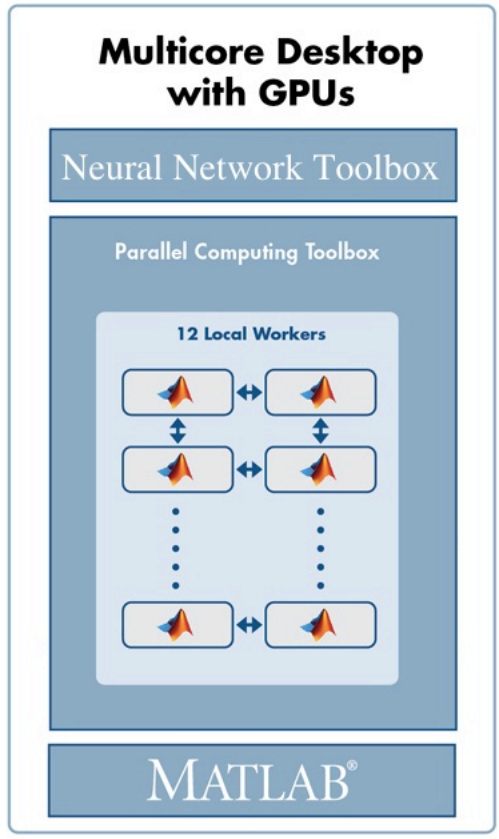
The previous way to indicate memory reduction was to set the `net.efficiency.memoryReduction` property before training:

```
net.efficiency.memoryReduction = N;
```

This continues to work in Version 8.0, but it is recommended that you update your code to use the `'reduction'` option for train and network simulation. Additional name-value pair arguments are the standard way to indicate calculation options.

Speedup of training and simulation with multicore processors and computer clusters using Parallel Computing Toolbox

Parallel Computing Toolbox allows Neural Network Toolbox simulation, and gradient and Jacobian calculations to be parallelized across multiple CPU cores, reducing calculation times. Parallelization splits the data among several workers. Results for the whole dataset are combined after all workers have completed their calculations.



Note that, during training, the calculation of network outputs, performance, gradient, and Jacobian calculations are parallelized, while the main training code remains on one worker.

To train a network on the `house_dataset` problem, introduced above, open a local MATLAB pool of workers, then call `train` and `sim` with the new `'useParallel'` option set to `'yes'`.

```
matlabpool open
numWorkers = matlabpool('size')
```

If calling `matlabpool` produces an error, it might be that Parallel Computing Toolbox is not available.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net = train(net,x,t,'useParallel','yes');
y = sim(net,'useParallel','yes');
```

On the sample system with a pool of four cores, typical speedups have been between 3x and 3.7x. Using more than four cores might produce faster speeds. For more information, see Parallel and GPU Computing.

GPU computing support for training and simulation on single and multiple GPUs using Parallel Computing Toolbox

Parallel Computing Toolbox allows Neural Network Toolbox simulation and training to be parallelized across the multiprocessors and cores of a graphics processing unit (GPU).

To train and simulate with a GPU set the `'useGPU'` option to `'yes'`. Use the gpuDevice command to get information on your GPU.

```
gpuInfo = gpuDevice
```

If calling `gpuDevice` produces an error, it might be that Parallel Computing Toolbox is not available.

Training on GPUs cannot be done with Jacobian algorithms, such as `trainlm` or `trainbr`, but it can be done with any of the gradient algorithms such as `trainscg`. If you do not change the training function, it will happen automatically.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net.trainFcn = 'trainscg';
net = train(net,x,t,'useGPU','yes');
y = sim(net,'useGPU','yes');
```

Speedups on the sample system with an nVidia GTX 470 GPU card have been between 3x and 7x, but might increase as GPUs continue to improve.

You can also use multiple GPUs. If you set both `'useParallel'` and `'useGPU'` to `'yes'`, any worker associated with a unique GPU will use that GPU, and other workers

will use their CPU core. It is not efficient to share GPUs between workers, as that would require them to perform their calculations in sequence instead of in parallel.

```
numWorkers = matlabpool('size')
numGPUs = gpuDeviceCount

[x,t] = house_dataset;
net = feedforwardnet(10);
net.trainFcn = 'trainscg';
net = train(net,x,t,'useParallel','yes','useGPU','yes');
y = sim(net,'useParallel','yes','useGPU','yes');
```

Tests with three GPU workers and one CPU worker on the sample system have seen 3x or higher speedup. Depending on the size of the problem, and how much it uses the capacity of each GPU, adding GPUs might increase speed or might simply increase the size of problem that can be run.

In some cases, training with both GPUs and CPUs can result in slower speeds than just training with the GPUs, because the CPUs might not keep up with the GPUs. In this case, set `'useGPU'` to `'only'` and only GPU workers will be used.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net = train(net,x,t,'useParallel','yes','useGPU','only');
y = sim(net,'useParallel','yes','useGPU','only');
```

For more information, see Parallel and GPU Computing.

Distributed training of large datasets on computer clusters using MATLAB Distributed Computing Server

Besides allowing load balancing, Composite data also allows datasets too large to fit within the RAM of a single computer to be distributed across the RAM of a cluster.

This is done by loading the Composite sequentially. For instance, here the sub-datasets are loaded from files as they are distributed:

```
Xc = Composite;
Tc = Composite;
for i=1:10
  data = load(['dataset' num2str(i)])
  Xc{i} = data.x;
  Tc{i} = data.t;
```

```
  clear data
end
```

This technique allows for training with datasets of any size, limited only by the available RAM across an entire cluster.

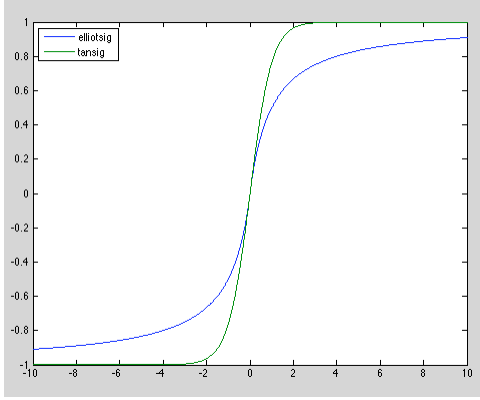
For more information, see Parallel and GPU Computing.

Elliot sigmoid transfer function for faster simulation

The new transfer function `elliotsig` calculates its output without using the `exp` function used by both `tansig` and `logsig`. This lets it execute much faster, especially on deployment hardware that might either not support `exp` or which implements it with software that takes many more execution cycles than simple arithmetic operations.

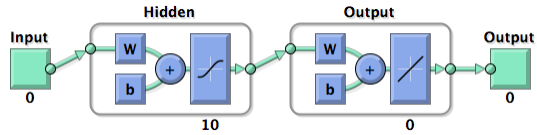
This example displays a plot of `elliotsig` alongside `tansig`:

```
n = -10:0.01:10;
a1 = elliotsig(n);
a2 = tansig(n);
h = plot(n,a1,n,a2);
legend(h,'ELLIOTSIG','TANSIG','Location','NorthWest')
```

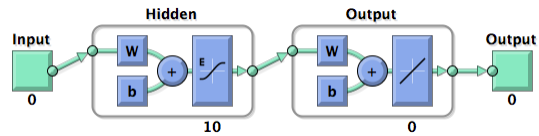


To set up a neural network to use the `elliotsig` transfer function, change each `tansig` layer's transfer function with its `transferFcn` property. For instance, here a network using `elliotsig` is created, viewed, trained, and simulated:

```
[x,t] = house_dataset;
net = feedforwardnet(10);
view(net) % View TANSIG network
```




```
net.layers{1}.transferFcn = 'elliotsig';
view(net) % View ELLIOTSIG network
```




```
net = train(net,x,t);
y = net(x)
```

The `elliotsig` transfer function might be even faster on an Intel® processor.

```
n = rand(1000,1000);
tic, for i=1:100, a = elliotsig(n); end, elliotsigTime = toc
tic, for i=1:100, a = tansig(n); end, tansigTime = toc
speedup = tansigTime / elliotsigTime
```

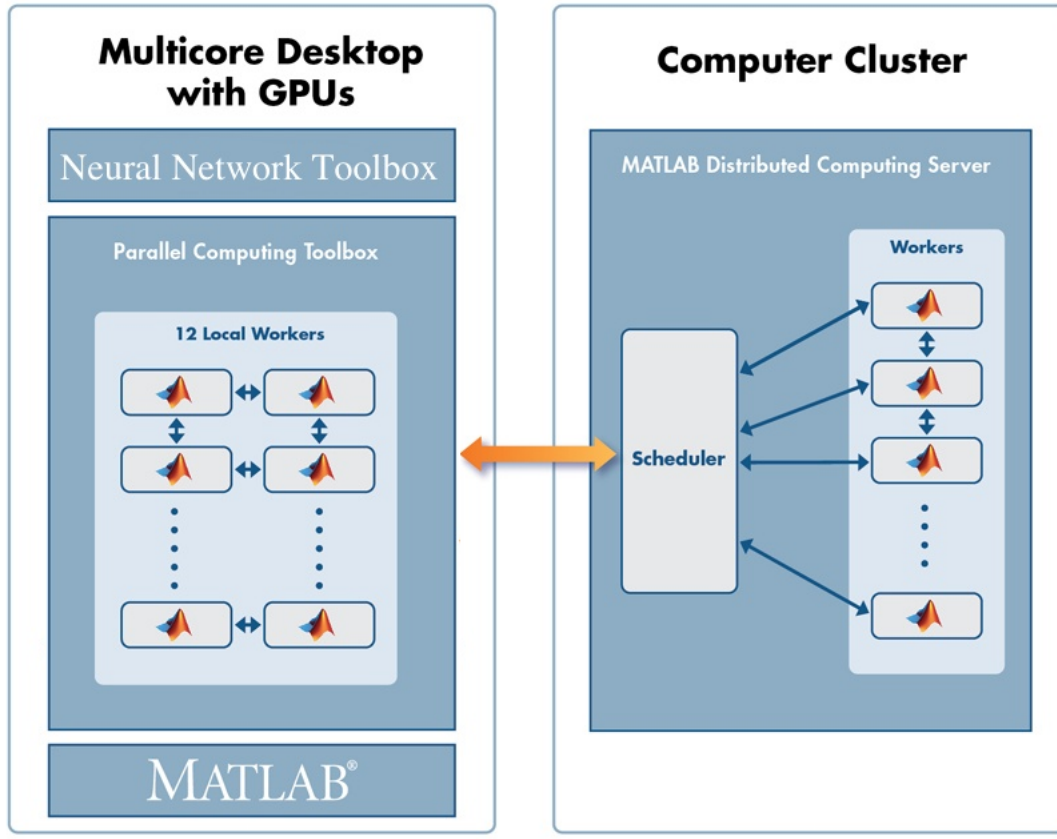
On one system the speedup was almost 3x.

However, because of the different shape, `elliotsig` might not result in faster training than `tansig`. It might require more training steps. For simulation, `elliotsig` is always faster.

For more information, see Fast Elliot Sigmoid.

Faster training and simulation with computer clusters using MATLAB Distributed Computing Server

If a MATLAB pool is opened using a cluster of computers, the previous parallel training and simulations happen across the CPU cores and GPUs of all the computers in the pool. For problems with hundreds of thousands or millions of samples, this might result in considerable speedup.



For more information, see Parallel and GPU Computing.

Load balancing parallel calculations

When training and simulating a network using the `'useParallel'` option, the dataset is automatically divided into equal parts across the workers. However, if different workers have different speed and memory limitations, it can be helpful to adjust the amount of data sent to each worker, so that the faster workers or those with more memory have proportionally more data.

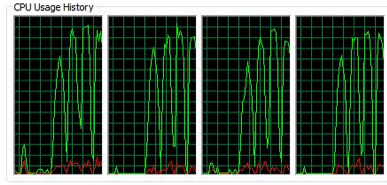
This is done using the Parallel Computing Toolbox function `Composite`. Composite data is data spread across a parallel pool of MATLAB workers.

For instance, if a parallel pool is open with four workers, data can be distributed as follows:

```
[x,t] = house_dataset;
Xc = Composite;
Tc = Composite;
Xc{1} = x(:,   1:150); % First  150 samples of x
Tc{1} = x(:,   1:150); % First  150 samples of t
Xc{2} = x(:, 151:300); % Second 150 samples of x
Tc{2} = x(:, 151:300); % Second 150 samples of t
Xc{3} = x(:, 301:403); % Third  103 samples of x
Tc{3} = x(:, 301:403); % Third  103 samples of t
Xc{4} = x(:, 404:506); % Fourth 103 samples of x
Tc{4} = x(:, 404:506); % Fourth 103 samples of t
```

When you call `train`, the `'useParallel'` option is not needed, because `train` automatically trains in parallel when using Composite data.

```
net = train(net,Xc,Tc);
```



If you want workers 1 and 2 to use GPU devices 1 and 2, while workers 3 and 4 use CPUs, set up data for workers 1 and 2 using `nndata2gpu` inside an `spmd` clause.

```
spmd
  if labindex <= 2
    Xc = nndata2gpu(Xc);
    Tc = nndata2gpu(Tc);
  end
end
```

The function `nndata2gpu` takes a neural network matrix or cell array time series data and converts it to a properly sized gpuArray on the worker's GPU. This involves transposing the matrices, padding the columns so their first elements are memory

aligned, and combining matrices, if the data was a cell array of matrices. To reverse process outputs returned after simulation with gpuArray data, use `gpu2nndata` to convert back to a regular matrix or a cell array of matrices.

As with `'useParallel'`, the data type removes the need to specify `'useGPU'`. Training and simulation automatically recognize that two of the workers have gpuArray data and employ their GPUs accordingly.

```
net = train(net,Xc,Tc);
```

This way, any variation in speed or memory limitations between workers can be accounted for by putting differing numbers of samples on those workers.

For more information, see Parallel and GPU Computing.

Summary and fallback rules of computing resources used from train and sim

The convention used for computing resources requested by options `'useParallel'` and `'useGPU'` is that if the resource is available it will be used. If it is not, calculations still occur accurately, but without that resource. Specifically:

1. If `'useParallel'` is set to `'yes'`, but no MATLAB pool is open, then computing occurs in the main MATLAB thread and is not distributed across workers.
2. If `'useGPU'` is set to `'yes'`, but there is not a suppported GPU device selected, then computing occurs on the CPU.
3. If `'useParallel'` and `'useGPU'` are set to `'yes'`, each worker uses a GPU if it is the first worker with a particular supported GPU selected, or uses a CPU core otherwise.
4. If `'useParallel'` is set to `'yes'` and `'useGPU'` is set to `'only'`, then only the first worker with a supported GPU is used, and other workers are not used. However, if no GPUs are available, calculations revert to parallel CPU cores.

Set the `'showResources'` option to `'yes'` to check what resources are actually being used, as opposed to requested for use, when training and simulating.

Example 11.1. Example: View computing resources

```
[x,t] = house_dataset;
net = feedforwardnet(10);
```

```
net2 = train(net,x,t,'showResources','yes');
y = net2(x,'showResources','yes');

Computing Resources:
MEX on PCWIN64

net2 = train(net,x,t,'useParallel','yes','showResources','yes');
y = net2(x,'useParallel','yes','showResources','yes');

Computing Resources:
  Worker 1 on Computer1, MEX on PCWIN64
  Worker 2 on Computer1, MEX on PCWIN64
  Worker 3 on Computer1, MEX on PCWIN64
  Worker 4 on Computer1, MEX on PCWIN64

net2 = train(net,x,t,'useGPU','yes','showResources','yes');
y = net2(x,'useGPU','yes','showResources','yes');

Computing Resources:
GPU device 1, TypeOfCard

net2 = train(net,x,t,'useParallel','yes','useGPU','yes',...
                                          'showResources','yes');
y = net2(x,'useParallel','yes','useGPU','yes','showResources','yes');

Computing Resources:
  Worker 1 on Computer1, GPU device 1, TypeOfCard
  Worker 2 on Computer1, GPU device 2, TypeOfCard
  Worker 3 on Computer1, MEX on PCWIN64
  Worker 4 on Computer1, MEX on PCWIN64

net2 = train(net,x,t,'useParallel','yes','useGPU','only',...
                                            'showResources','yes');
y = net2(x,'useParallel','yes','useGPU','only','showResources','yes');

Computing Resources:
  Worker 1 on Computer1, GPU device 1, TypeOfCard
  Worker 2 on Computer1, GPU device 2, TypeOfCard
```

Updated code organization

The code organization for data processing, weight, net input, transfer, performance, distance and training functions are updated. Custom functions of these kinds need to be updated to the new organization.

In Version 8.0 the related functions for neural network processing are in package folders, so each local function has its own file.

For instance, in Version 7.0 the function `tansig` contained a large switch statement and several local functions. In Version 8.0 there is a root function `tansig`, along with several package functions in the folder `/toolbox/nnet/nnet/nntransfer/+tansig/`.

```
+tansig/activeInputRange.m
+tansig/apply.m
+tansig/backprop.m
+tansig/da_dn.m
+tansig/discontinuity.m
+tansig/forwardprop.m
+tansig/isScalar.m
+tansig/name.m
+tansig/outputRange.m
+tansig/parameterInfo.m
+tansig/simulinkParameters.m
+tansig/type.m
```

Each transfer function has its own package with the same set of package functions. For lists of processing, weight, net input, transfer, performance, and distance functions, each of which has its own package, type the following:

```
help nnprocess
help nnweight
help nnnetinput
help nntransfer
help nnperformance
help nndistance
```

The calling interfaces for training functions are updated for the new calculation modes and parallel support. Normally, training functions would not be called directly, but indirectly by train, so this is unlikely to require any code changes.

Compatibility Considerations

Due to the new package organization for processing, weight, net input, transfer, performance and distance functions, any custom functions of these types will need to be updated to conform to this new package system before they will work with Version 8.0.

See the main functions and package functions for `mapminmax`, `dotprod`, `netsum`, `tansig`, `mse`, and `dist` for examples of this new organization. Any of these functions and its package functions may be used as a template for new or updated custom functions.

Due to the new calling interfaces for training functions, any custom backpropagation training function will need to be updated to work with Version 8.0. See `trainlm` and `trainscg` for examples that can be used as templates for any new or updated custom training function.

R2012a

Version: 7.0.3

Bug Fixes

R2011b

Version: 7.0.2

Bug Fixes

R2011a

Version: 7.0.1

Bug Fixes

R2010b

Version: 7.0

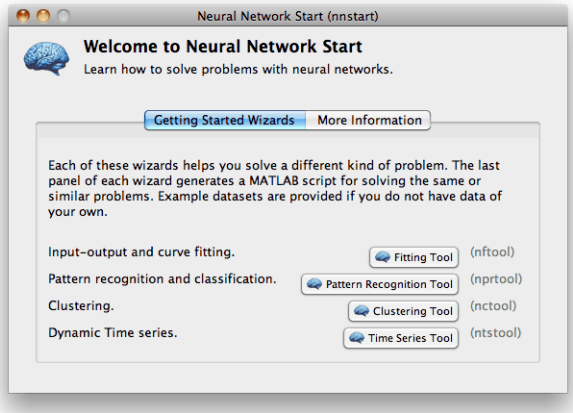
New Features

Bug Fixes

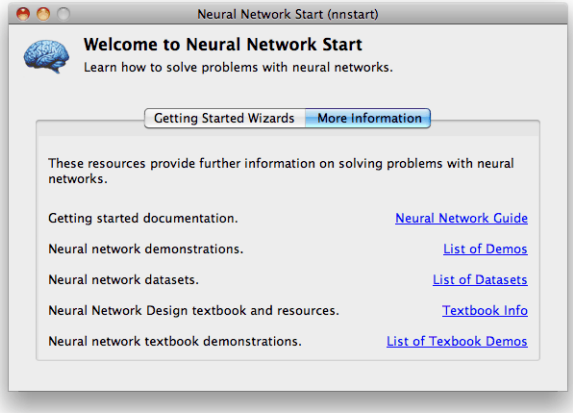
Compatibility Considerations

New Neural Network Start GUI

The new `nnstart` function opens a GUI that provides links to new and existing Neural Network Toolbox GUIs and other resources. The first panel of the GUI opens four "getting started" wizards.

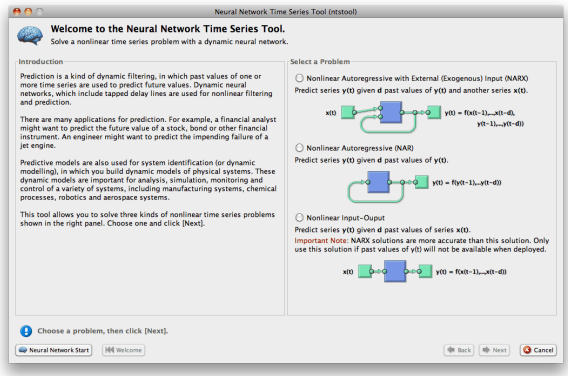


The second panel provides links to other toolbox starting points.

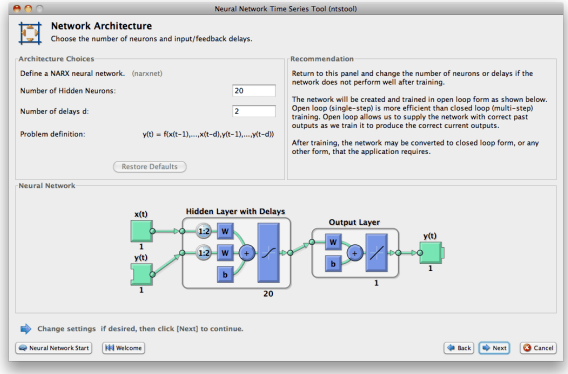


New Time Series GUI and Tools

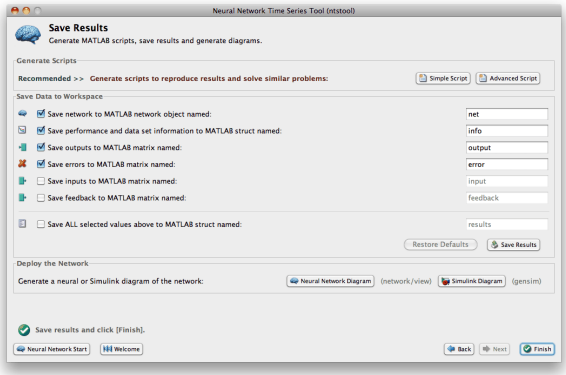
The new `ntstool` function opens a wizard GUI that allows time series problems to be solved with three kinds of neural networks: NARX networks (neural auto-regressive with external input), NAR networks (neural auto-regressive), and time delay neural networks. It follows a similar format to the neural fitting (`nftool`), clustering (`nctool`), and pattern recognition (`nprtool`) tools.

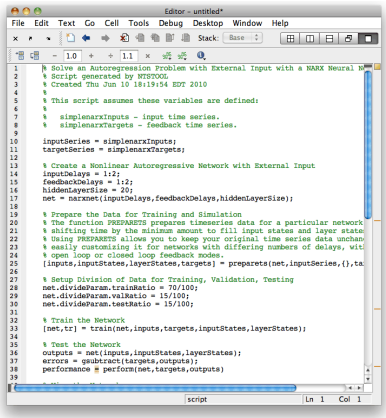


Network diagrams shown in the Neural Time Series Tool, Neural Training Tool, and with the `view(net)` command, have been improved to show tap delay lines in front of weights, the sizes of inputs, layers and outputs, and the time relationship of inputs and outputs. Open loop feedback outputs and inputs are indicated with matching tab and indents in their respective blocks.



The Save Results panel of the Neural Network Time Series Tool allows you to generate both a Simple Script, which demonstrates how to get the same results as were obtained with the wizard, and an Advanced Script, which provides an introduction to more advanced techniques.

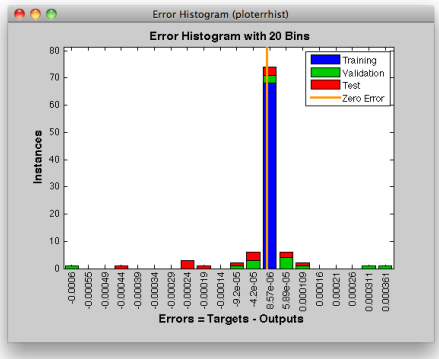




The Train Network panel of the Neural Network Time Series Tool introduces four new plots, which you can also access from the Network Training Tool and the command line.

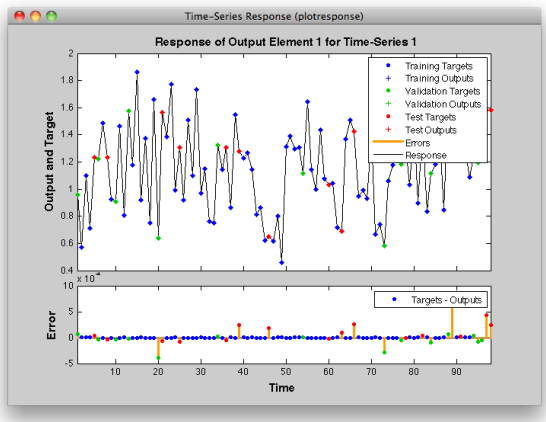
The error histogram of any static or dynamic network can be plotted.

```
plotresponse(errors)
```



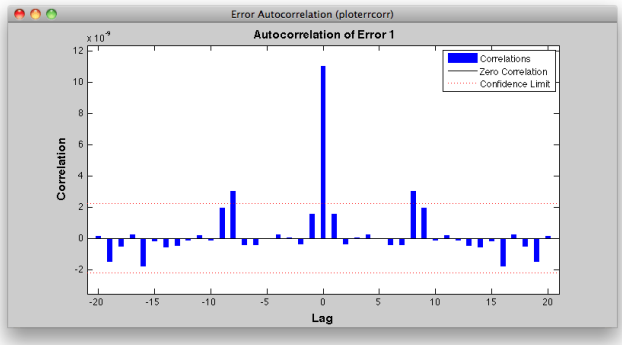
The dynamic response can be plotted, with colors indicating how targets were assigned to training, validation and test sets across timesteps. (Dividing data by timesteps and other criteria, in addition to by sample, is a new feature described in “New Time Series Validation” on page 15-9.)

```
plotresponse(targets,outputs)
```



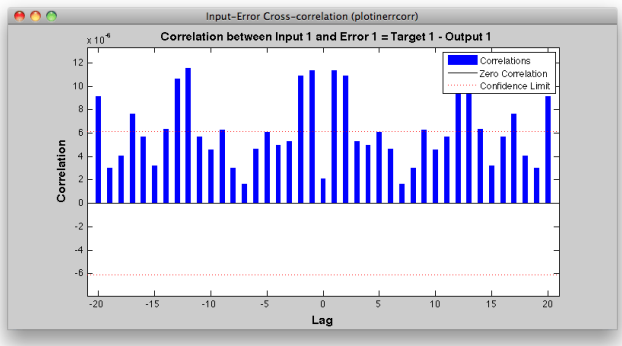
The autocorrelation of error across varying lag times can be plotted.

```
ploterrcorr(errors)
```



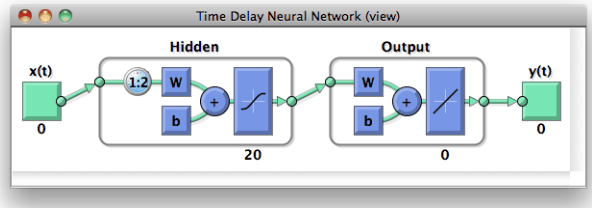
The input-to-error correlation can also be plotted for varying lags.

```
plotinerrcorr(inputs,errors)
```



Simpler time series neural network creation is provided for NARX and time-delay networks, and a new function creates NAR networks. All the network diagrams shown here are generated with the command `view(net)`.

```
net = narxnet(inputDelays, feedbackDelays, hiddenSizes,
feedbackMode, trainingFcn
net = narnet(feedbackDelays, hiddenSizes, feedbackMode,
trainingFcn)
net = timedelaynet(inputDelays, hiddenSizes, trainingFcn)
```



Several new data sets provide sample problems that can be solved with these networks. These data sets are also available within the `ntstool` GUI and the command line.

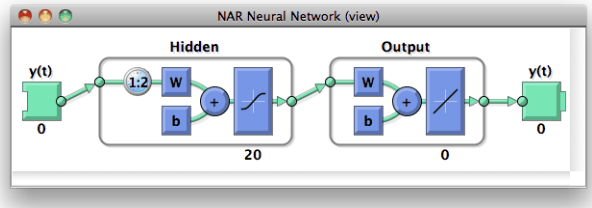
```
[x, t] = simpleseries_dataset;
[x, t] = simplenarx_dataset;
[x, t] = exchanger_dataset;
[x, t] = maglev_dataset;
[x, t] = ph_dataset;
[x, t] = pollution_dataset;
[x, t] = refmodel_dataset;
[x, t] = robotarm_dataset;
[x, t] = valve_dataset;
```

The `preparets` function formats input and target time series for time series networks, by shifting the inputs and targets as needed to fill initial input and layer delay states. This function simplifies what is normally a tricky data preparation step that must be customized for details of each kind of network and its number of delays.

```
[x, t] = simplenarx_dataset;
net = narxnet(1:2, 1:2, 10);
[xs, xi, ai, ts] = preparets(net, x, {}, t);
net = train(net, xs, ts, xi, ai);
y = net(xs, xi, ai)
```

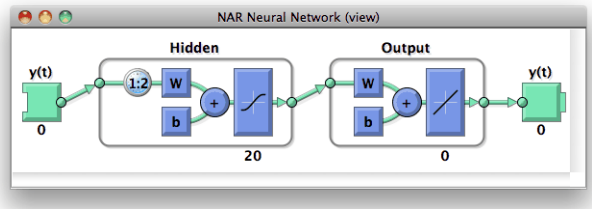
The output-to-input feedback of NARX and NAR networks (or custom time series network with output-to-input feedback loops) can be converted between open- and closed-loop modes using the two new functions `closeloop` and `openloop`.

```
net = narxnet(1:2, 1:2, 10);
net = closeloop(net)
net = openloop(net)
```



The total delay through a network can be adjusted with the two new functions `removedelay` and `adddelay`. Removing a delay from a NARX network which has a minimum input and feedback delay of 1, so that it now has a minimum delay of 0, allows the network to predict the next target value a timestep ahead of when that value is expected.

```
net = removedelay(net)
net = adddelay(net)
```



The new function `catsamples` allows you to combine multiple time series into a single neural network data variable. This is useful for creating input and target data from multiple input and target time series.

```
x = catsamples(x1, x2, x3);
t = catsamples(t1, t2, t3);
```

In the case where the time series are not the same length, the shorter time series can be padded with NaN values. This will indicate “don't care” or equivalently “don't know” input and targets, and will have no effect during simulation and training.

```
x = catsamples(x1, x2, x3, 'pad')
t = catsamples(t1, t2, t3, 'pad')
```

Alternatively, the shorter series can be padded with any other value, such as zero.

```
x = catsamples(x1, x2, x3, 'pad', 0)
```

There are many other new and updated functions for handling neural network data, which make it easier to manipulate neural network time series data.

```
help nndatafun
```

New Time Series Validation

Normally during training, a data set's targets are divided up by sample into training, validation and test sets. This allows the validation set to stop training at a point of optimal generalization, and the test set to provide an independent measure of the network's accuracy. This mode of dividing up data is now indicated with a new property:

```
net.divideMode = 'sample'
```

However, many time series problems involve only a single time series. In order to support validation you can set the new property to divide data up by timestep. This is the default setting for NARXNET and other time series networks.

```
net.divideMode = 'time'
```

This property can be set manually, and can be used to specify dividing up of targets across both sample and timestep, by all target values (i.e., across sample, timestep, and output element), or not to perform data division at all.

```
net.divideMode = 'sampletime'
net.divideMode = 'all'
net.divideMode = 'none'
```

New Time Series Properties

Time series feedback can also be controlled manually with new network properties that represent output-to-input feedback in open- or closed-loop modes. For open-loop feedback from an output from layer `i` back to input `j`, set these properties as follows:

```
net.inputs{j}.feedbackOutput = i
net.outputs{i}.feedbackInput = j
net.outputs{i}.feedbackMode = 'open'
```

When the feedback mode of the output is set to `'closed'`, the properties change to reflect that the output-to-input feedback is now implemented with internal feedback by removing input `j` from the network, and having output properties as follows:

```
net.outputs{i}.feedbackInput = [];
net.outputs{i}.feedbackMode = 'closed'
```

Another output property keeps track of the proper closed-loop delay, when a network is in open-loop mode. Normally this property has this setting:

```
net.outputs{i}.feedbackDelay = 0
```

However, if a delay is removed from the network, it is updated to 1, to indicate that the network's output is actually one timestep ahead of its inputs, and must be delayed by 1 if it is to be converted to closed-loop form.

```
net.outputs{i}.feedbackDelay = 1
```

New Flexible Error Weighting and Performance

Performance functions have a new argument list that supports error weights for indicating which target values are more important than others. The `train` function also supports error weights.

```
net = train(net, x, t, xi, ai, ew)
perf = mse(net, x, t, ew)
```

You can define error weights by sample, output element, time step, or network output:

```
ew = [1.0 0.5 0.7 0.2];      % Weighting errors across 4 samples
ew = [0.1; 0.5; 1.0];        % ... across 3 output elements
ew = {0.1 0.2 0.3 0.5 1.0};  % ... across 5 timesteps
ew = {1.0; 0.5};             % ... across 2 network outputs
```

These can also be defined across any combination. For example, weighting error across two time series (i.e., two samples) over four timesteps:

```
ew = {[0.5 0.4], [0.3 0.5], [1.0 1.0], [0.7 0.5]};
```

In the general case, error weights can have exactly the same dimension as targets, where each target has an associated error weight.

Some performance functions are now obsolete, as their functionality has been implemented as options within the four remaining performance functions: `mse`, `mae`, `sse`, and `sae`.

The regularization implemented in `msereg` and `msnereg` is now implemented with a performance property supported by all four remaining performance functions.

```
% Any value between the default 0 and 1.
net.performParam.regularization
```

The error normalization implemented in `msne` and `msnereg` is now implemented with a normalization property.

```
% Either 'normalized', 'percent', or the default 'none'.
net.performParam.normalization
```

A third performance parameter indicates whether error weighting is applied to square errors (the default for `mse` and `sse`) or the absolute errors (`mae` and `sae`).

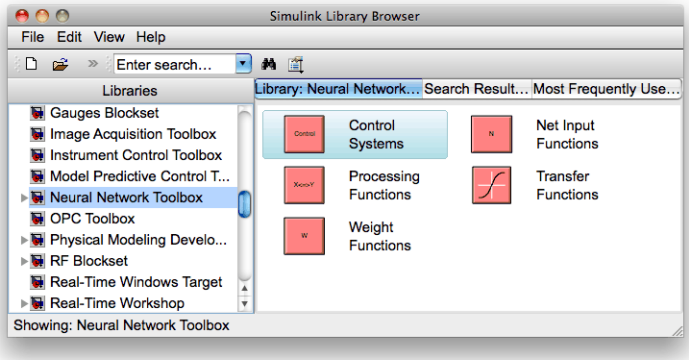
```
net.performParam.squaredWeighting  % true or false
```

Compatibility Considerations

The old performance functions and old performance arguments lists continue to work as before, but are no longer recommended.

New Real Time Workshop and Improved Simulink Support

Neural network Simulink blocks now compile with Real Time Workshop® and are compatible with Rapid Accelerator mode.



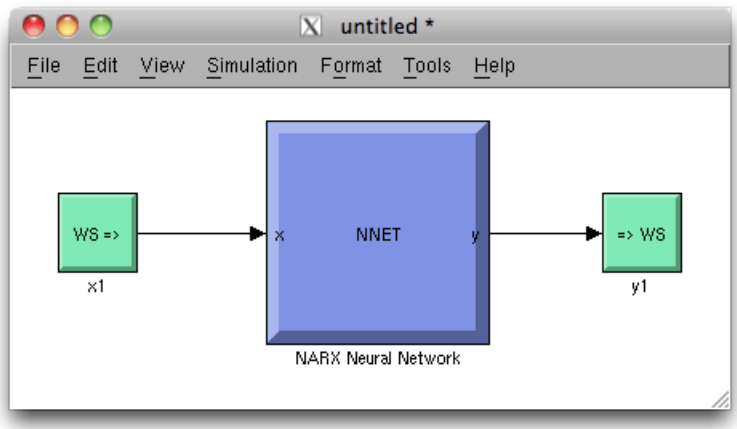
`gensim` has new options for generating neural network systems in Simulink.

```
Name - the system name
SampleTime - the sample time
```

```
InputMode - either port, workspace, constant, or none.
OutputMode - either display, port, workspace, scope, or none
SolverMode - either default or discrete
```

For instance, here a NARX network is created and set up in MATLAB to use workspace inputs and outputs.

```
[x, t] = simplenarx_dataset;
net = narxnet(1:2, 1:2, 10);
[xs, xi, ai, ts] = preparets(net, x, {}, t);
net = train(net, xs, ts, xi, ai);
net = closeloop(net);
[sysName, netName] = gensim(net, 'InputMode', 'workspace', ...
           'OutputMode', 'workspace', 'SolverMode', 'discrete');
```



Simulink neural network blocks now allow initial conditions for input and layer delays to be set directly by double-clicking the neural network block. `setsiminit` and `getsiminit` provide command-line control for setting and getting input and layer delays for a neural network Simulink block.

```
setsiminit(sysName, netName, net, xi, ai);
```

New Documentation Organization and Hyperlinks

The User's Guide has been rearranged to better focus on the workflow of practical applications. The Getting Started section has been expanded.

References to functions throughout the online documentation and command-line help now link directly to their function pages.

```
help feedforwardnet
```

The command-line output of neural network objects now contains hyperlinks to documentation. For instance, here a feed-forward network is created and displayed. Its command-line output contains links to network properties, function reference pages, and parameter information.

```
net = feedforwardnet(10);
```

Subobjects of the network, such as inputs, layers, outputs, biases, weights, and parameter lists also display with links.

```
net.inputs{1}
net.layers{1}
net.outputs{2}
net.biases{1}
net.inputWeights{1, 1}
net.trainParam
```

The training tool `nntraintool` and the wizard GUIs `nftool`, `nprtool`, `nctool`, and `ntstool`, provide numerous hyperlinks to documentation.

New Derivative Functions and Property

New functions give convenient access to error gradient (of performance with respect to weights and biases) and Jacobian (of error with respect to weights and biases) calculated by various means.

```
staticderiv - Backpropagation for static networks
bttderiv - Backpropagation through time
fpderiv - Forward propagation
num2deriv - Two-point numerical approximation
num5deriv - Five-point numerical approximation
defaultderiv - Chooses recommended derivative function for the network
```

For instance, here you can calculate the error gradient for a newly created and configured feedforward network.

```
net = feedforwardnet(10);
[x, t] = simplefit_dataset;
```

```
net = configure(net, x, t);
d = staticderiv('dperf_dwb', net, x, t)
```

Improved Network Creation

New network creation functions have clearer names, no longer need example data, and have argument lists reduced to only the arguments recommended for most applications. All arguments have defaults, so you can create simple networks by calling network functions without any arguments. New networks are also more memory efficient, as they no longer need to store sample input and target data for proper configuration of input and output processing settings.

```
% New function
net = feedforwardnet(hiddenSizes, trainingFcn)

% Old function
net = newff(x,t,hiddenSizes, transferFcns, trainingFcn, ...
      learningFcn, performanceFcn, inputProcessingFcns, ...
      outputProcessingFcns, dataDivisionFcn)
```

The new functions (and the old functions they replace) are:

`feedforwardnet` (`newff`)
`cascadeforwardnet` (`newcf`)
`competlayer` (`newc`)
`distdelaynet` (`newdtdnn`)
`elmannet` (`newelm`)
`fitnet` (`newfit`)
`layrecnet` (`newlrn`)
`linearlayer` (`newlin`)
`lvqnet` (`newlvq`)
`narxnet` (`newnarx`, `newnarxsp`)
`patternnet` (`newpr`)
`perceptron` (`newp`)
`selforgmap` (`newsom`)
`timedelaynet` (`newtdnn`)

The network's inputs and outputs are created with size zero, then configured for data when `train` is called or by optionally calling the new function `configure`.

```
net = configure(net, x, t)
```

Unconfigured networks can be saved and reused by configuring them for many different problems. `unconfigure` sets a configured network's inputs and outputs to zero, in a network which can later be configured for other data.

```
net = unconfigure(net)
```

Compatibility Considerations

Old functions continue working as before, but are no longer recommended.

Improved GUIs

The neural fitting `nftool`, pattern recognition `nprtool`, and clustering `nctool` GUIs have been updated with links back to the `nnstart` GUI. They give the option of generating either simple or advanced scripts in their last panel. They also confirm with you when closing, if a script has not been generated, or the results not yet saved.

Improved Memory Efficiency

Memory reduction, the technique of splitting calculations up in time to reduce memory requirements, has been implemented across all training algorithms for both gradient and network simulation calculations. Previously it was only supported for gradient calculations with `trainlm` and `trainbr`.

To set the memory reduction level, use this new property. The default is 1, for no memory reduction. Setting it to 2 or higher splits the calculations into that many parts.

```
net.efficiency.memoryReduction
```

Compatibility Considerations

The `trainlm` and `trainbr` training parameter MEM_REDUC is now obsolete. References to it will need to be updated. Code referring to it will generate a warning.

Improved Data Sets

All data sets in the toolbox now have help, including example solutions, and can be accessed as functions:

```
help simplefit_dataset
[x, t] = simplefit_dataset;
```

See help for a full list of sample data sets:

```
help nndatasets
```

Updated Argument Lists

The argument lists for the following types of functions, which are not generally called directly, have been updated.

The argument list for training functions, such as `trainlm`, `traingd`, etc., have been updated to match `train`. The argument list for the adapt function `adaptwb` has been updated. The argument list for the layer and network initialization functions, `initlay`, `initnw`, and `initwb` have been updated.

Compatibility Considerations

Any custom functions of these types, or code which calls these functions manually, will need to be updated.

R2010a

Version: 6.0.4

Bug Fixes

R2009b

Version: 6.0.3

Bug Fixes

R2009a

Version: 6.0.2

Bug Fixes

R2008b

Version: 6.0.1

Bug Fixes

R2008a

Version: 6.0

New Features

Bug Fixes

Compatibility Considerations

New Training GUI with Animated Plotting Functions

Training networks with the `train` function now automatically opens a window that shows the network diagram, training algorithm names, and training status information.

The window also includes buttons for plots associated with the network being trained. These buttons launch the plots during or after training. If the plots are open during training, they update every epoch, resulting in animations that make understanding network performance much easier.

The training window can be opened and closed at the command line as follows:

```
nntraintool
nntraintool('close')
```

Two plotting functions associated with the most networks are:

- `plotperform`—Plot performance.
- `plottrainstate`—Plot training state.

Compatibility Considerations

To turn off the new training window and display command-line output (which was the default display in previous versions), use these two training parameters:

```
net.trainParam.showWindow = false;
net.trainParam.showCommandLine = true;
```

New Pattern Recognition Network, Plotting, and Analysis GUI

The `nprtool` function opens a GUI wizard that guides you to a neural network solution for pattern recognition problems. Users can define their own problems or use one of the new data sets provided.

The `newpr` function creates a pattern recognition network at the command line. Pattern recognition networks are feed-forward networks that solve problems with Boolean or 1-of-*N* targets and have confusion (`plotconfusion`) and receiver operating characteristic (`plotroc`) plots associated with them.

The new `confusion` function calculates the true/false, positive/negative results from comparing network output classification with target classes.

New Clustering Training, Initialization, and Plotting GUI

The `nctool` function opens a GUI wizard that guides you to a self-organizing map solution for clustering problems. Users can define their own problem or use one of the new data sets provided.

The `initsompc` function initializes the weights of self-organizing map layers to accelerate training. The `learnsomb` function implements batch training of SOMs that is orders of magnitude faster than incremental training. The `newsom` function now creates a SOM network using these faster algorithms.

Several new plotting functions are associated with self-organizing maps:

- `plotsomhits`—Plot self-organizing map input hits.
- `plotsomnc`—Plot self-organizing map neighbor connections.
- `plotsomnd`—Plot self-organizing map neighbor distances.
- `plotsomplanes`—Plot self-organizing map input weight planes.
- `plotsompos`—Plot self-organizing map weight positions.
- `plotsomtop`—Plot self-organizing map topology.

Compatibility Considerations

You can call the `newsom` function using conventions from earlier versions of the toolbox, but using its new calling conventions gives you faster results.

New Network Diagram Viewer and Improved Diagram Look

The new neural network diagrams support arbitrarily connected network architectures and have an improved layout. Their visual clarity has been improved with color and shading.

Network diagrams appear in all the Neural Network Toolbox graphical interfaces. In addition, you can open a network diagram viewer of any network from the command line by typing

```
view(net)
```

New Fitting Network, Plots and Updated Fitting GUI

The `newfit` function creates a fitting network that consistes of a feed-forward backpropagation network with the fitting plot (`plotfit`) associated with it.

The `nftool` wizard has been updated to use `newfit`, for simpler operation, to include the new network diagrams, and to include sample data sets. It now allows a Simulink block version of the trained network to be generated from the final results panel.

Compatibility Considerations

The code generated by `nftool` is different the code generated in previous versions. However, the code generated by earlier versions still operates correctly.

R2007b

Version: 5.1

New Features

Bug Fixes

Compatibility Considerations

Simplified Syntax for Network-Creation Functions

The following network-creation functions have new input arguments to simplify the network creation process:

- `newcf`
- `newff`
- `newdtdnn`
- `newelm`
- `newfftd`
- `newlin`
- `newlrn`
- `newnarx`
- `newnarxsp`

For detailed information about each function, see the corresponding reference pages.

Changes to the syntax of network-creation functions have the following benefits:

- You can now specify input and target data values directly. In the previous release, you specified input ranges and the size of the output layer instead.
- The new syntax automates preprocessing, data division, and postprocessing of data.

For example, to create a two-layer feed-forward network with 20 neurons in its hidden layer for a given a matrix of input vectors `p` and target vectors `t`, you can now use `newff` with the following arguments:

```
net = newff(p,t,20);
```

This command also sets properties of the network such that the functions `sim` and `train` automatically preprocess inputs and targets, and postprocess outputs.

In the previous release, you had to use the following three commands to create the same network:

```
pr = minmax(p);
s2 = size(t,1);
net = newff(pr,[20 s2]);
```

Compatibility Considerations

Your existing code still works but might produce a warning that you are using obsolete syntax.

Automated Data Preprocessing and Postprocessing During Network Creation

Automated data preprocessing and postprocessing occur during network creation in the Network/Data Manager GUI (`nntool`), Neural Network Fitting Tool GUI (`nftool`), and at the command line.

At the command line, the new syntax for using network-creation functions, automates preprocessing, postprocessing, and data-division operations.

For example, the following code returns a network that automatically preprocesses the inputs and targets and postprocesses the outputs:

```
net = newff(p,t,20);
net = train(net,p,t);
y = sim(net,p);
```

To create the same network in a previous release, you used the following longer code:

```
[p1,ps1] = removeconstantrows(p);
[p2,ps2] = mapminmax(p1);
[t1,ts1] = mapminmax(t);
pr = minmax(p2);
s2 = size(t1,1);
net = newff(pr,[20 s2]);
net = train(net,p2,t1);
y1 = sim(net,p2)
y = mapminmax('reverse',y1,ts1);
```

Default Processing Settings

The default input `processFcns` functions returned with a new network are, as follows:

```
net.inputs{1}.processFcns = ...
              {'fixunknowns','removeconstantrows', 'mapminmax'}
```

These three processing functions perform the following operations, respectively:

- `fixunknowns`—Encode unknown or missing values (represented by `NaN`) using numerical values that the network can accept.
- `removeconstantrows`—Remove rows that have constant values across all samples.
- `mapminmax`—Map the minimum and maximum values of each row to the interval `[-1 1]`.

The elements of `processParams` are set to the default values of the `fixunknowns`, `removeconstantrows`, and `mapminmax` functions.

The default output `processFcns` functions returned with a new network include the following:

```
net.outputs{2}.processFcns = {'removeconstantrows','mapminmax'}
```

These defaults process outputs by removing rows with constant values across all samples and mapping the values to the interval `[-1 1]`.

`sim` and `train` automatically process inputs and targets using the input and output processing functions, respectively. `sim` and `train` also reverse-process network outputs as specified by the output processing functions.

For more information about processing input, target, and output data, see "Multilayer Networks and Backpropagation Training" in the Neural Network Toolbox User's Guide.

Changing Default Input Processing Functions

You can change the default processing functions either by specifying optional processing function arguments with the network-creation function, or by changing the value of `processFcns` after creating your network.

You can also modify the default parameters for each processing function by changing the elements of the `processParams` properties.

After you create a network object (`net`), you can use the following input properties to view and modify the automatic processing settings:

- `net.inputs{1}.exampleInput`—Matrix of example input vectors
- `net.inputs{1}.processFcns`—Cell array of processing function names
- `net.inputs{1}.processParams`—Cell array of processing parameters

The following input properties are automatically set and you cannot change them:

- `net.inputs{1}.processSettings`—Cell array of processing settings
- `net.inputs{1}.processedRange`—Ranges of example input vectors after processing
- `net.inputs{1}.processedSize`—Number of input elements after processing

Changing Default Output Processing Functions

After you create a network object (`net`), you can use the following output properties to view and modify the automatic processing settings:

- `net.outputs{2}.exampleOutput`—Matrix of example output vectors
- `net.outputs{2}.processFcns`—Cell array of processing function names
- `net.outputs{2}.processParams`—Cell array of processing parameters

> **Note** These output properties require a network that has the output layer as the second layer.

The following new output properties are automatically set and you cannot change them:

- `net.outputs{2}.processSettings`—Cell array of processing settings
- `net.outputs{2}.processedRange`—Ranges of example output vectors after processing
- `net.outputs{2}.processedSize`—Number of input elements after processing

Automated Data Division During Network Creation

When training with supervised training functions, such as the Levenberg-Marquardt backpropagation (the default for feed-forward networks), you can supply three sets of input and target data. The first data set trains the network, the second data set stops training when generalization begins to suffer, and the third data set provides an independent measure of network performance.

Automated data division occurs during network creation in the Network/Data Manager GUI, Neural Network Fitting Tool GUI, and at the command line.

At the command line, to create and train a network with early stopping that uses 20% of samples for validation and 20% for testing, you can use the following code:

```
net = newff(p,t,20);
net = train(net,p,t);
```

Previously, you entered the following code to accomplish the same result:

```
pr = minmax(p);
s2 = size(t,1);
net = newff(pr,[20 s2]);
[trainV,validateV,testV] = dividevec(p,t,0.2,0.2);
[net,tr] = train(net,trainV.P,trainV.T,[],[],validateV,testV);
```

For more information about data division, see “Multilayer Networks and Backpropagation Training” in the Neural Network Toolbox User's Guide.

New Data Division Functions

The following are new data division functions:

- `dividerand`—Divide vectors using random indices.
- `divideblock`—Divide vectors in three blocks of indices.
- `divideint`—Divide vectors with interleaved indices.
- `divideind`—Divide vectors according to supplied indices.

Default Data Division Settings

Network creation functions return the following default data division properties:

- `net.divideFcn = 'dividerand'`
- `net.divedeParam.trainRatio = 0.6;`
- `net.divideParam.valRatio = 0.2;`
- `net.divideParam.testRatio = 0.2;`

Calling `train` on the network object net divided the set of input and target vectors into three sets, such that 60% of the vectors are used for training, 20% for validation, and 20% for independent testing.

Changing Default Data Division Settings

You can override default data division settings by either supplying the optional data division argument for a network-creation function, or by changing the corresponding property values after creating the network.

After creating a network, you can view and modify the data division behavior using the following new network properties:

- `net.divideFcn`—Name of the division function
- `net.divideParam`—Parameters for the division function

New Simulink Blocks for Data Preprocessing

New blocks for data processing and reverse processing are available. For more information, see “Processing Blocks” in the Neural Network Toolbox User's Guide.

The function `gensim` now generates neural networks in Simulink that use the new processing blocks.

Properties for Targets Now Defined by Properties for Outputs

The properties for targets are now defined by the properties for outputs. Use the following properties to get and set the output and target properties of your network:

- `net.numOutputs`—The number of outputs and targets
- `net.outputConnect`—Indicates which layers have outputs and targets
- `net.outputs`—Cell array of output subobjects defining each output and its target

Compatibility Considerations

Several properties are now obsolete, as described in the following table. Use the new properties instead.

Recommended Property	Obsolete Property
`net.numOutputs`	`net.numTargets`
`net.outputConnect`	`net.targetConnect`
`net.outputs`	`net.targets`

R2007a

Version: 5.0.2

No New Features or Changes

R2006b

Version: 5.0.1

No New Features or Changes

R2006a

Version: 5.0

New Features

Compatibility Considerations

Dynamic Neural Networks

Version 5.0 now supports these types of dynamic neural networks:

Time-Delay Neural Network

Both focused and distributed time-delay neural networks are now supported. Continue to use the `newfftd` function to create focused time-delay neural networks. To create distributed time-delay neural networks, use the `newdtdnn` function.

Nonlinear Autoregressive Network (NARX)

To create parallel NARX configurations, use the `newnarx` function. To create series-parallel NARX networks, use the `newnarxsp` function. The `sp2narx` function lets you convert NARX networks from series-parallel to parallel configuration, which is useful for training.

Layer Recurrent Network (LRN)

Use the `newlrn` function to create LRN networks. LRN networks are useful for solving some of the more difficult problems in filtering and modeling applications.

Custom Networks

The training functions in Neural Network Toolbox are enhanced to let you train arbitrary custom dynamic networks that model complex dynamic systems. For more information about working with these networks, see the Neural Network Toolbox documentation.

Wizard for Fitting Data

The new Neural Network Fitting Tool (`nftool`) is now available to fit your data using a neural network. The Neural Network Fitting Tool is designed as a wizard and walks you through the data-fitting process step by step.

To open the Neural Network Fitting Tool, type the following at the MATLAB prompt:

```
nftool
```

Data Preprocessing and Postprocessing

Version 5.0 provides the following new data preprocessing and postprocessing functionality:

dividevec Automatically Splits Data

The `dividevec` function facilitates dividing your data into three distinct sets to be used for training, cross validation, and testing, respectively. Previously, you had to split the data manually.

fixunknowns Encodes Missing Data

The `fixunknowns` function encodes missing values in your data so that they can be processed in a meaningful and consistent way during network training. To reverse this preprocessing operation and return the data to its original state, call `fixunknowns` again with `'reverse'` as the first argument.

removeconstantrows Handles Constant Values

`removeconstantrows` is a new helper function that processes matrices by removing rows with constant values.

mapminmax, mapstd, and processpca Are New

The `mapminmax`, `mapstd`, and `processpca` functions are new and perform data preprocessing and postprocessing operations.

Compatibility Considerations

Several functions are now obsolete, as described in the following table. Use the new functions instead.

New Function	Obsolete Functions

`mapminmax`	`premnmx`
	`postmnmx`
	`tramnmx`

`mapstd`	`prestd`
	`poststd`
	`trastd`

New Function	Obsolete Functions
`processpca`	`prepca` `trapca`

Each new function is more efficient than its obsolete predecessors because it accomplishes both preprocessing and postprocessing of the data. For example, previously you used `premnmx` to process a matrix, and then `postmnmx` to return the data to its original state. In this release, you accomplish both operations using `mapminmax`; to return the data to its original state, you call `mapminmax` again with `'reverse'` as the first argument:

```
mapminmax('reverse',Y,PS)
```

Derivative Functions Are Obsolete

The following derivative functions are now obsolete:

```
ddotprod
dhardlim
dhardlms
dlogsig
dmae
dmse
dmsereg
dnetprod
dnetsum
dposlin
dpurelin
dradbas
dsatlin
dsatlins
dsse
dtansig
dtribas
```

Each derivative function is named by prefixing a `d` to the corresponding function name. For example, `sse` calculates the network performance function and `dsse` calculated the derivative of the network performance function.

Compatibility Considerations

To calculate a derivative in this version, you must pass a derivative argument to the function. For example, to calculate the derivative of a hyperbolic tangent sigmoid transfer function `A` with respect to `N`, use this syntax:

```
A = tansig(N,FP)
dA_dN = tansig('dn',N,A,FP)
```

Here, the argument `'dn'` requests the derivative to be calculated.

R14SP3

Version: 4.0.6

No New Features or Changes